Appeared in Proc. UIST'00
(ACM Symp. on User Interface Software and Technology, San Diego, CA, November 5-8,2000, 238-248)

Dynamic Space Management for User Interfaces

Blaine A. Bell *Steven K. Feiner*
Columbia University
Department of Computer Science
500 West 120th St., 450 CS Building, New York, NY 10027
+1 212 939-7000
E-mail: {bab22,feiner}@cs.columbia.edu

ABSTRACT
We present a general approach to the dynamic representation of 2D space that is well suited for user-interface layout. We partition space into two distinct categories: full and empty. The user can explicitly specify a set of possibly overlapping upright rectangles that represent the objects of interest. These full-space rectangles are processed by the system to create a representation of the remaining empty space. This representation makes it easy for users to develop customized spatial allocation strategies that avoid overlapping the full-space rectangles. We describe the representation; provide efficient incremental algorithms for adding and deleting full-space rectangles, and for querying the empty-space representation; and show several allocation strategies that the representation makes possible. We present two testbed applications that incorporate an implementation of the algorithm; one shows the utility of our representation for window management tasks; the other applies it to the layout of components in a 3D user interface, based on the upright 2D bounding boxes of their projections.

KEYWORDS: Spatial data structures, user interface design, geometric modeling, display layout, space allocation, window management, overlap avoidance.

1. INTRODUCTION

Graphics systems commonly provide a representation of the space occupied by the elements of a scene. These representations are often hierarchical, and are typified by the window trees of 2D window systems [14], and the bounding volume hierarchies of 3D graphics packages and renderers [4, 10]. A variety of operations, such as rendering, clipping, and collision detection, depend on these representations for efficiency.

There are many situations in which we would like to allocate space for an object, while avoiding intersecting other objects that have already been allocated. For example, we may wish to place a window in a window system such that it meets criteria for position, size, or aspect ratio, yet does not overlap existing windows. When we turn to the space representations typically used in interactive graphics, we find that it is easy to determine whether a conflict has occurred, either by a linear comparison with a list of allocated objects, or by more efficient approaches provided by a hierarchy of bounds or an interval tree [13]. However, these representations do not make it easy to determine how to modify a desired placement or select a new one to avoid the conflict, or even whether it is possible to position an object anywhere in the scene without overlap. The result is a plethora of applications that adopt a simplistic approach to allocating space. These are typified by window managers that position newly created windows at arbitrary positions, or that apply simple tiling or cascading strategies on demand, but which do not maintain them as users move and resize objects. Consequently, users often need to modify object geometry by hand. This can be especially tedious when large numbers of objects or multiple applications are involved.

To address these problems, we have developed an efficient approach for representing, building, and querying *empty space*—the dual of the *full space* occupied by objects of interest to the user.

2. APPROACH

For simplicity and efficiency, we support floating-point axis-aligned rectangles in a rectangular workspace. The user can incrementally add or delete full-space rectangles. These full-space rectangles may represent the exact dimensions of objects (e.g., the rectangular windows of a 2D GUI) or approximations (e.g., the upright extents of the 2D projections of 3D objects). At any time, the user can query the empty-space representation, which is automatically maintained by the system. The empty-space representation consists of largest empty-space rectangles, each of whose height and width is as large as it can be without overlapping any full-space rectangle. That is, each of the largest empty-space rectangles is bounded on the left,

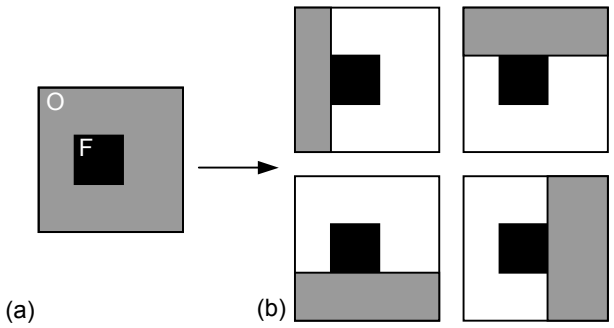


Figure 1: (a) Adding a black full-space rectangle *F* to an empty scene, represented by a light gray single largest empty-space rectangle *O,* (b) creates four new light gray largest empty-space rectangles.

right, top, and bottom by an edge of a full-space rectangle or an edge of the workspace. For example, Figure 1 shows the four possible largest empty-space rectangles surrounding a single full-space rectangle. Note that the largest empty-space rectangles do not partition the empty space, but can overlap, as in Figure 1(b).

This representation is well suited for determining the placement of upright rectangular objects. Any rectangle that can be placed within the empty space will be completely enclosed by at least one of the largest empty-space rectangles. Finding a largest empty-space rectangle that satisfies simple geometric constraints on position, width, height, or aspect ratio, requires only simple comparisons with the desired full-space rectangle to be added. For this approach to be successful, the empty-space representation must be computed automatically, offering interactive performance to applications that need it.

In the remainder of this paper, we first describe previous related work in Section 3. Next, in Section 4, we present efficient incremental algorithms for maintaining the empty-space representation as the user adds and deletes full-space rectangles, and for querying the empty-space representation. Then, we describe our implementation and present timings for operations in Section 5. In Section 6, we provide examples from two prototype applications that make use of our implementation: a simple window manager testbed and a knowledge-based 3D presentation designer. Finally, we present our conclusions and plans for future work in Section 7.

3. PREVIOUS WORK

A number of spatial representation schemes have been developed that explicitly represent empty space. For example, quadtrees and octrees [13] provide 2D and 3D discrete approximations of full and empty space using a hierarchy of axis-aligned cells. Corner stitching [11], the exact space representation for axis-aligned rectangles used in the MAGIC VLSI layout system [12], supports efficient queries of full and empty spaces adjacent to a given full or empty space. However, the basic structure of the corner-stitching algorithm tiles the plane with full-space and empty-space rectangles, without supporting overlapping rectangles. Overlapping full-space rectangles have been addressed in corner stitching by using multiple overlapping layers (each individually corner-stitched), or by decomposing overlapped rectangles into a set of non-overlapping rectangles before adding them. All these representations partition the empty space into non-overlapping pieces. Thus, they do not make it easy for the user to find empty space that can accommodate a desired rectangle that is larger than any of these individual partitions.

Bernard and Jacquenet [3] use the same empty-space representation that we do, presenting an incremental algorithm for adding full-space rectangles. However, they do not support overlapping full-space rectangles or deletion of full-space rectangles. Both of these capabilities are important for general interactive applications, such as those of Section 6.

Several tiled window managers have used empty-space representations to allocate space when non-overlapping windows are created, moved, or resized. Many tiled window managers are restricted to hierarchical layouts (e.g., [15, 8]) in which windows (i.e., full-space rectangles) are allocated in a strict hierarchy by recursively partitioning the workspace horizontally or vertically at each subdivision. Thus, no full-space or empty-space rectangle can straddle partitions. Although it is straightforward to query the empty-space representation in a hierarchical system, this is possible only because of the restricted geometry that prohibits combining empty-space rectangles. In contrast, the Siemens RTL/RTL and constraint-based RTL/CRTL tiled window managers [5] support nonhierarchical layouts, relying on corner stitching. However, adjacent empty-space partitions in corner stitching must be combined in both dimensions to determine whether they can accommodate a window of a specified width or height.

4. ALGORITHMS FOR DYNAMIC SPACE MANAGEMENT

Our space manager provides efficient algorithms for incrementally adding and deleting full-space rectangles, and for querying the empty-space representation that it automatically maintains.

4.1 Adding a Full-Space Rectangle

Our algorithm for incrementally adding a full-space rectangle is similar to that of [3] with an important difference needed to support overlapping rectangles. Given an existing list of full-space rectangles with an empty-space representation, when a full-space rectangle *F* is added, the set of largest empty-space rectangles needs to be modified. Only those largest empty-space rectangles that intersect *F* will be affected, since they are the only rectangles whose bounds can change because of the addition.

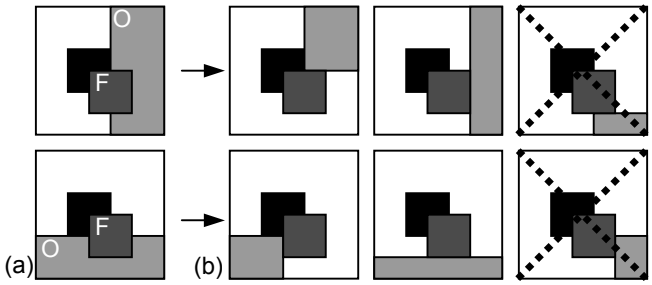


Figure 2: Adding a full-space rectangle to Figure 1. (a) Full-space rectangle *F* (dark gray) reduces two largest empty-space rectangles *O* (light gray). (b) Empty-space rectangles (light gray) created. Dashed "X"s indicate new empty-space rectangles that are not largest empty-space rectangles.

As in [3], the fundamental operation for adding a full-space rectangle is *reduction*, the creation of new, smaller empty-space rectangles from an existing largest empty-space rectangle *O* that intersects full-space rectangle *F*. For each edge *e* of *F* that intersects *O* and is not collinear with any edge of *O*, a new empty-space rectangle is created. This new empty-space rectangle is bounded by *e* and the three other edges of *O*; thus it corresponds to the portion of *O* that is on the side of *e* outside of *F*. For example, as shown at the top left of Figure 1(b), if the left edge of *F* intersects *O*, then *O* can be reduced to a rectangle whose left, top, and bottom edges are inherited from *O*, and whose right edge is the left edge of *F*.

Rectangle *O* can be reduced by rectangle *F* to create between zero and four new, smaller rectangles: one for each edge of *F* that intersects the interior of *O*. For example, Figure 1 shows the four empty-space rectangles created by reducing outer rectangle *O* by the black rectangle *F* that is fully contained in *O* and touches none of *O*'s edges. In contrast, *F* cannot reduce *O* at all if *O* is fully contained in *F* and touches none of *F*'s edges.

The rectangles created by reducing *O* by *F* represent the empty-space remaining in *O* after the addition of *F*. Thus, to maintain the empty-space representation, we must use *F* to reduce each largest empty-space rectangle *O* that it intersects. Each of the original largest empty-space rectangles that intersect *F* must be removed from the list of largest empty-space rectangles. However, we cannot simply replace the rectangles that are removed by the rectangles

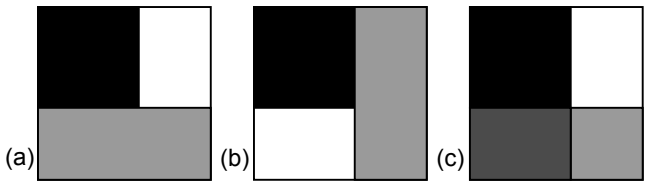


Figure 3: (a–b) Black full-space rectangle generates two light gray largest empty-space rectangles. (c) Adding dark gray full-space rectangle to left side of largest empty space in (a) generates empty space on right that must be compared with largest empty space in (b) to determine that the new empty space is not a largest empty space.

created by reducing them—some of the new reduced rectangles may be contained within others. For example, Figure 2 shows the six empty-space rectangles created by adding overlapping full-space rectangle *F* to Figure 1, reducing the two largest empty-space rectangles that overlap *F* in Figure 2(a). Two of these six new empty-space rectangles are contained inside the other four (and are "X"ed out in Figure 2b). Therefore, we must add only these four largest empty-space rectangles to the list of largest empty-space rectangles.

A new empty-space rectangle adjacent to edge *e* of *F* only needs to be tested for containment against other empty-space rectangles adjacent to *e*. However, it is important to note that it must be tested against all such empty-space rectangles, including those largest empty-space rectangles that share only one edge *e* with *F*, and which therefore cannot be reduced by *F*! This is shown in Figure 3, where adding the dark gray rectangle in part (c) reduces the light gray largest empty-space rectangle in part (a) to create a single new empty-space rectangle on its right in part (c). However, this new empty-space rectangle is wholly contained within the light gray largest empty-space rectangle in part (b), and therefore must not be added to the list of largest empty-space rectangles.

Figure 4 shows the pseudocode for the incremental add.

```
SpaceManager.addFullRectangle(Rectangle F){
  fullSpaceList.add(F)
  CList = existing rectangles in largestEmptySpaceList
    that intersect or are adjacent to F
  Vector possibleLESList[4] = new Vector[4]
  Vector adjacentLESList[4] = new Vector[4]
  for each empty space O in CList {
    if (O adjacent to any side e of F and external to F)
      adjacentLESList[e].add(O)
    else {
      largestEmptySpaceList.delete(O)
      if (O.minx < F.minx)      /* Figure 1(b), top left          */
        possibleLESList[LEFT].add(
          Rectangle(O.minX,F.minX,O.minY,O.maxY))
      if (O.maxX > F.maxX)   /* Figure 1(b), bottom right   */
        possibleLESList[RIGHT].add(
          Rectangle(F.maxX,O.maxX,O.minY,O.maxY))
      if (O.minY < F.minY)     /* Figure 1(b), bottom left    */
        possibleLESList[BOTTOM].add(
          Rectangle(O.minX,O.maxX,O.minY,F.minY))
      if (O.maxY > F.maxY)   /* Figure 1(b), top right         */
        possibleLESList[TOP].add(
          Rectangle(O.minX,O.maxX,F.maxY,O.maxY))
    }
  }
  for each Direction D in {LEFT,RIGHT,BOTTOM,TOP}{
    for each empty space P in possibleLESList[D] {
      if (P not enclosed by any space in
          adjacentLESList[D] or any other space in
          possibleLESList[D])
        largestEmptySpaceList.add(P)
    }
  }
}
```

Figure 4: Incremental addition of a full-space rectangle

First, the set of largest empty-space rectangles that intersect *F* or are adjacent to it are computed using a window query (see Section 4.3). Next, the rectangles of this set are compared with *F*. Each rectangle that is adjacent to a side of *F* and external to *F* is added to a list of largest empty-space rectangles adjacent to that side. All other rectangles in the set are deleted from the set of largest empty-space rectangles and are reduced by *F* to create a separate list of new empty-space rectangles for each edge of *F*. Then each edge's possible empty-space rectangle list is culled to eliminate members that are not largest empty-space rectangles, and the surviving members are added to the list of largest empty-space rectangles.

Our addition operation differs from that of Bernard and Jacquenet in its ability to handle overlapping rectangles. We use one query operation to obtain the set of relevant overlapping or adjacent largest empty-space rectangles and process each against the newly added rectangle *F*. In contrast, Bernard and Jacquenet use four query operations (one for each edge of *F*) to find all largest empty-space rectangles intersected by each edge and then reduce them by that edge. Checking only edge intersection, rather than rectangle intersection, does not handle overlapping rectangles properly. For example, suppose that *F* fully covers a largest empty-space rectangle *O,* and *F*'s edges lie wholly outside *O*. Rectangle *O* should be deleted, but this will not occur when *F*'s edges are used because *O* is not intersected by any of *F*'s edges. Bernard and Jacquenet also do not take into account the need for comparison with adjacent largest empty-space rectangles, demonstrated in Figure 3. This results in spurious additions to the list of largest empty-spaces.

4.2 Deleting a Full-Space Rectangle

Deleting a full-space rectangle *F* is essentially the inverse of adding it. We need to modify the largest empty-space list to include all largest empty-space rectangles that intersect *F*'s area after it is deleted. At first it might seem that we would only have to reconstruct the largest empty-space rectangles destroyed when *F* was added, by 1D expansion (the inverse of reduction) of the empty-space rectangles created by reducing each largest empty-space rectangle *O* that intersected *F*. Unfortunately, this is not possible, as shown in Figure 5, because the empty-space rectangles that would need to be expanded may have been discarded because they were not largest empty-space rectangles. Furthermore, unless full-space rectangles are deleted in the exact inverse of the order in which they were created, the largest empty-space rectangles that will intersect *F*'s area after it is deleted may be different than those that intersected its area before it was added. In fact, full-space rectangles intersected by or wholly contained in *F* may define largest empty-space rectangles unrelated to those present before *F*'s deletion.

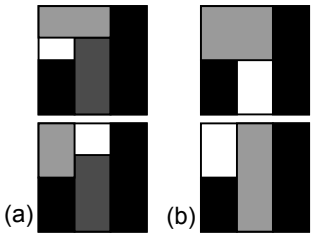


Figure 5: (a) Two black full-space rectangles and one dark gray full-space rectangle and the two light gray largest empty-space rectangles they define. (b) Deleting dark gray full-space rectangle should yield this new set of light gray largest empty-space rectangles. Note that these largest empty-space rectangles cannot be obtained by expanding each light gray rectangle in (a) in one dimension after deleting the dark gray rectangle.

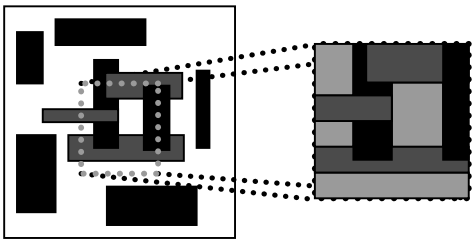


Figure 6: The deletion of an overlapping full-space rectangle, shown as gray dotted outline (left). Recursive space manager is created to process the area formerly occupied by the deleted rectangle, to determine the four light gray largest empty-space rectangles within the area (right).

To delete *F*, we must find all the largest empty-space rectangles that intersect *F*'s area after deletion. We compute these by first creating two lists: the largest empty-space rectangles external to and adjacent to *F*, and the empty-space rectangles wholly contained in *F*'s area after its deletion. The list of adjacent external largest empty-space rectangles is obtained with a window query. To find the list of empty-space rectangles within *F*'s area, we recursively invoke our space manager for the area bounded by *F*, adding to it all full-space rectangles intersecting *F*, not including *F* itself. The empty-space rectangles found within *F*'s area are thus the largest empty-space rectangles relative to *F,* as shown in Figure 6.

```
Method combineSpaces(Rectangle R1,Rectangle R2) {
  if (isAdjacentInX(R1,R2))
    return (Rectangle(MIN(R1.minX,R2.minX),
                      MAX(R1.minY,R2.minY),
                      MAX(R1.maxX,R2.maxX),
                      MIN(R1.maxY,R2.maxY)))
  else if (isAdjacentInY(R1,R2))
    return (Rectangle(MAX(R1.minX,R2.minX),
                      MIN(R1.minY,R2.minY),
                      MIN(R1.maxX,R2.maxX),
                      MAX(R1.maxY,R2.maxY)))
}
```

(a)

(b)

Figure 7: (a) Combining adjacent spaces. (b) Two empty spaces (outlined) that are adjacent in *Y* are combined to create a third (light gray) empty space

```
SpaceManager.deleteFullRectangle(Rectangle F){
   remove F from fullSpaceList
   create new Space Manager S to represent area of F
   get all full-space rectangles that intersect F
      and add them to S
   adjacentEmptySpaceList  = all largest empty-space
      rectangles adjacent to and external to F
   /* Note: these are the only largest empty spaces external
      to F that need to be processed in this algorithm */
   /* possibleLESList is a temporary list of
      empty spaces that could possibly be largest empty
      spaces; initialize it to the empty spaces in S */
   possibleLESList = S.largestEmptySpaceList
   for each edge e in (LEFT,RIGHT,BOTTOM,TOP) {
      adjList = spaces in possibleLESList adjacent to edge e of F
      for each empty space R in adjList {
         for each empty space P in adjacentEmptySpaceList  {
            if (P adjacent to edge e of R and edge e of F)
               possibleLESList .add(combineSpaces (R, P))
               /* see Figure 7 for function combineSpaces    */
         }
         if (any rectangle in possibleLESList  encloses R)
            remove R from possibleLESList
      }
   }
   for each empty space R in adjacentEmptySpaceList {
      if (R enclosed by a rectangle in possibleLESList)
         largestEmptySpaceList.delete(R)
   }
   for each empty space R in possibleLESList {
      if (R not enclosed by any other space in
            possibleLESList)
         largestEmptySpaceList.add(R)
   }
}
```

Figure 8: Incremental deletion of a full-space rectangle

For each empty-space rectangle found within F's area, we determine the set of edges of F to which it is adjacent. An empty-space rectangle inside F's area that is adjacent to no edges of F can be added to the list of largest empty-space rectangles without any further processing.

An internal empty-space rectangle that is adjacent to one or more edges of F may be able to be combined with one or more of the external largest empty-space rectangles adjacent to F to create new largest empty-space rectangles. Therefore, we keep a list of rectangles that are possibly largest empty-space rectangles. At the end of the algorithm, we add a rectangle to the largest empty rectangle list only if it exists in the possible list and is not enclosed by any other rectangle in the list. We must also delete the empty-space rectangles that were previously adjacent to F if they are contained within any of the largest empty-space rectangles added.

We combine empty-space rectangles by noting that only adjacent rectangles can be merged. If two rectangles are adjacent in one dimension, then they can be used to spawn a new rectangle whose extent in that dimension is the union of the two rectangles' extents in that dimension, and whose extent in the other dimension is the intersection of the two

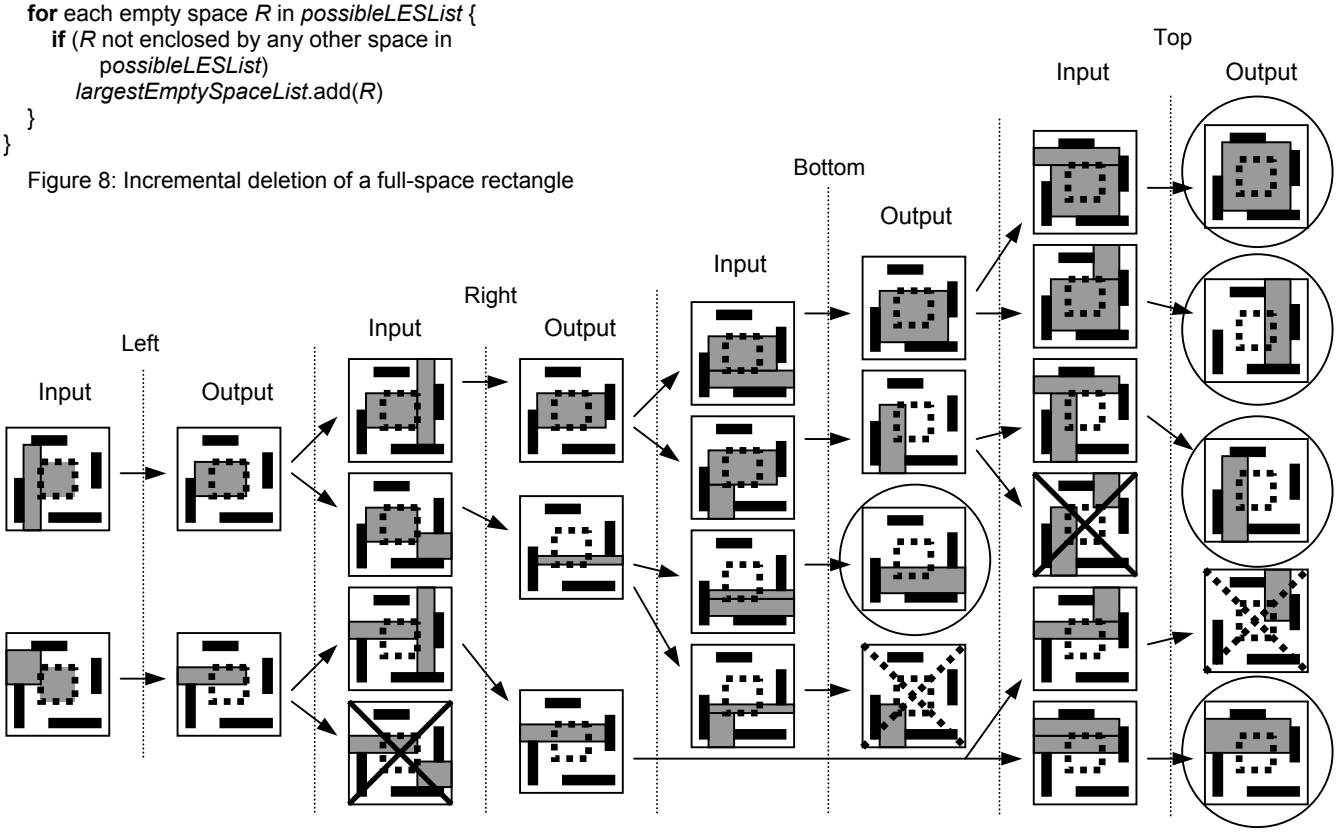


Figure 9: Deletion of a non-overlapping full-space rectangle (shown as heavy dashed black outline). Each box in an edge's input column shows a pair of empty-space rectangles that are candidates for combination to create an empty-space rectangle in that edge's output column. Circled boxes contain completely processed empty-space rectangles that will be added to the space manager's list of largest empty-space rectangles when the deletion is completed. Boxes covered by a solid "X" contain pairs of spaces that cannot be combined. Boxes covered by a dashed "X" contain completely processed empty-space rectangles that are subsets of other empty spaces computed in this process, and which will be discarded.

rectangles' extents in the other dimension, as shown in Figure 7. If we iteratively combine each empty-space rectangle R within F with all those adjacent to it outside of F, we will create the set of largest empty-space rectangles. Nowick (personal communication) has pointed out that this is analogous to the consensus theorem used in digital logic to find prime implicants [9].

Figure 8 shows the pseudocode for the incremental delete. We accomplish this merging operation by treating the edges of F in sequence, as depicted in Figure 9. For each edge e of F, each internal empty-space rectangle R adjacent to e is merged with the external largest empty-space rectangles adjacent to R. The resulting merged rectangles are added to a merge list, as are all of an edge's internal empty-space rectangles that are not wholly contained in any of the merged rectangles. When the next edge e of F is processed, the empty-space rectangles on the merge list that are also adjacent to e must be merged with all other empty-space rectangles that are adjacent to e. Taking into account the edges of F shared by empty-space rectangles avoids the brute force approach of merging explicitly every combination of internal and external empty-space rectangles adjacent to all edges of F.

4.3 Querying the Representation

The space manager maintains separate data structures for the full-space and largest empty-space rectangles. To optimize the worst-case time and space for querying rectangles within a given set of bounds in X and Y, we use a 2D interval tree for each set of rectangles, requiring worst-case $O(\log^2 N + M)$ time for a query and $O(N \log N)$ space for the data structure, where N is the number of rectangles in the data structure and M is the number of rectangles returned by the query [13].

5. IMPLEMENTATION

We have implemented the space manager in Java 1.2. Performance depends on the distribution of the rectangles, especially within the trajectories of moving full-space rectangles, and benefits greatly from using the deletion algorithm, instead of recomputing the entire space representation when an object is to be deleted. In general, when moving one full-space rectangle with more than three others in the scene, or moving two full-space rectangles with more than six others in the scene, we have found that it is better to delete and add the moving rectangles from the space manager, rather than to add all full-space rectangles from scratch.

Figure 10 compares the time it takes to do a move operation by deleting and adding the moving full-space rectangle (*Delete/Add*) vs. adding all full-space rectangles into an empty space manager (*Add All*). This figure graphs elapsed time for randomly generated sets of full-space rectangles on a 733 MHz Intel Pentium III processor with 256MB RAM, running Windows 2000. There are considerable differences, (e.g., for 100 full-space rectangles

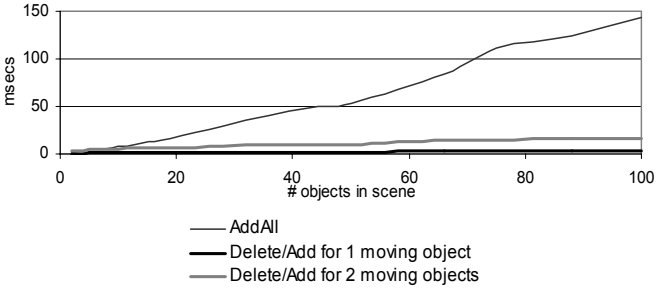


Figure 10: Performance comparison of *Add All* vs. *Delete/Add* operation durations for different scenes.

we recorded ~3 msec to delete and add one more full-space rectangle, vs. 143 msec to add all the full-space rectangles).

As the number of moving objects and total scene objects increase, both the *Add All* and *Delete/Add* method can become quite slow. This is especially noticeable if many objects are moved continuously, with space allocation performed at each frame. To address this problem, we have implemented an efficient undo capability that makes it possible to undo the most recent add operations quickly. In addition to the regular add operation, we provide an *undoable add* that saves the largest empty-space rectangles that it removes, and marks the new largest empty-space rectangles that it creates. An undoable add can be undone by having the space manager remove the largest empty-space rectangles it created and reinstantiate the largest empty-space rectangles that it removed. This undoable add makes it possible to animate a selected group of objects more quickly than by repeatedly deleting and adding them. To accomplish this, the programmer first deletes the objects to be animated and adds them back with new locations or sizes by using undoable adds. Then, for each frame in the animation, the previous frame's undoable adds are undone and the modified objects are added back with undoable adds. Figure 11 compares the performance of *Delete/Add* with *undoable add*.

6. EXAMPLES

We have applied the space manager to two prototype applications to explore the kind of functionality that it makes possible.

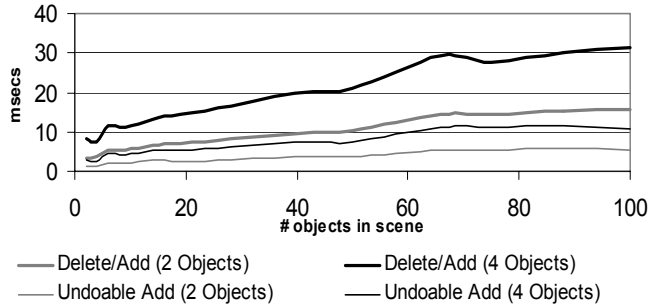


Figure 11: Performance comparison of *Delete/Add* vs. *undoable add* operation durations for different scenes.

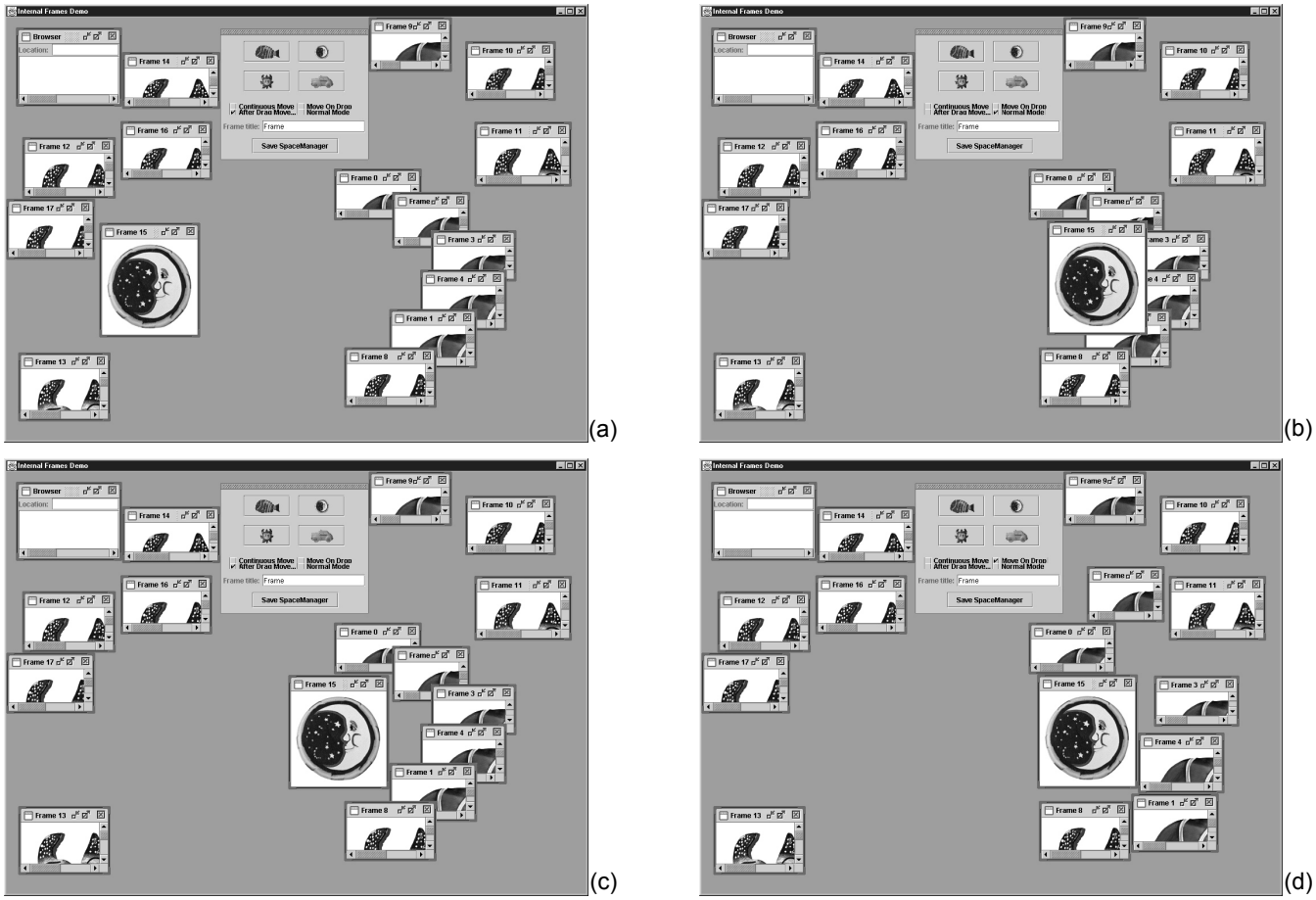


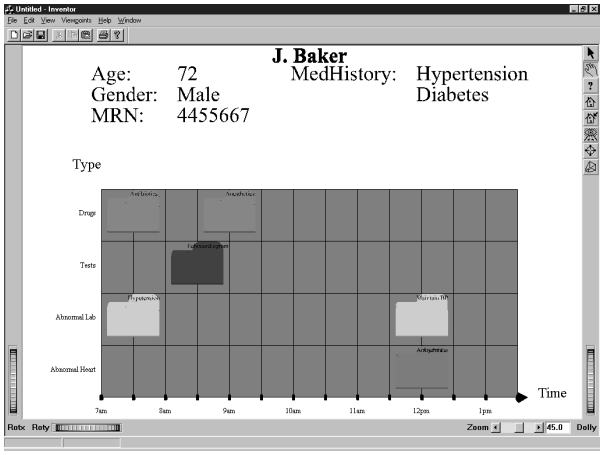
Figure 12: Window manager testbed. (a) Dragging the window (the moon) at the lower left, (b) conventionally results in placement where window is dropped. (c) *Overlap-avoidance drag* interpolates the dragged window to a sufficiently large empty-space rectangle closest to its original destination. (d) Alternative overlap-avoidance approach interpolates all windows overlapped by dragged window to sufficiently large empty spaces closest to their original locations.

6.1 Window Manager Testbed

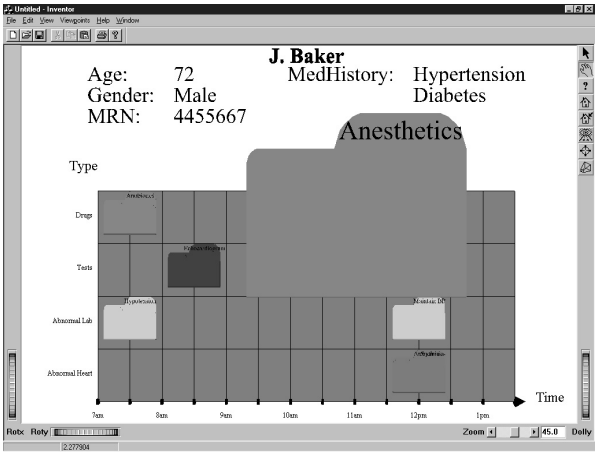
We have modified the Java InternalFrameDemo [7] by using our space manager to create a testbed for window manager tasks. By making it easy to examine the list of largest empty-space rectangles, we can provide some of the benefits of a tiling window manager, such as the lack of window obscuration, within the framework of a desktop overlapping window manager.

Figure 12(a–b) shows the screen before and after a conventional window dragging operation. We explored two new kinds of *overlap-avoidance* dragging. In both approaches, all windows are added to the space representation as they are created. In one approach (Figure 12c), a window is deleted from the representation when it is dragged. When the user releases the window at a new location, the window manager determines whether it overlaps any existing windows and queries the largest empty-space representation to find the closest empty space that can contain the dragged window. If that space is within a parameterized distance of the window's current location, the window is added to the representation and smoothly moved there by the system; otherwise it is added at its original destination. This allows the user to move a window quickly and sloppily, overlapping it with adjacent ones at the new location, knowing that if the window is dragged to a location that is sufficiently near a large enough empty space, it will move to the closest such space to avoid overlap.

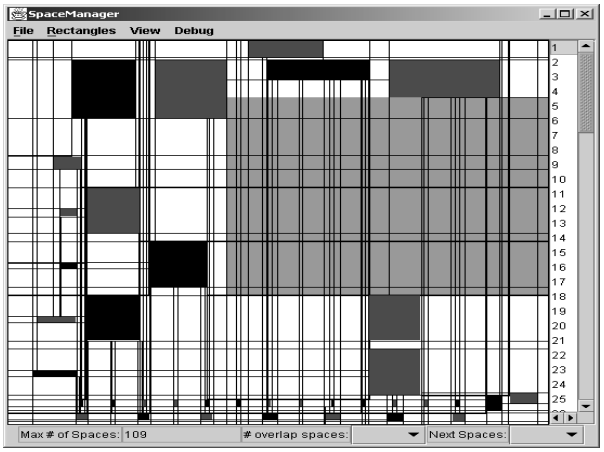
An alternative approach (Figure 12d) examines the multiple windows that are overlapped by the moved window, deletes each of them from the space manager, and moves them to new destinations to avoid overlap. In order to calculate the destinations, the space manager is used to find the closest empty space for each individual rectangle. However, calculating these destinations separately causes this greedy algorithm to give priority to windows that are processed first, and can fragment the space, resulting in a missed solution. (Alternatively, we could try all possible processing orders, and compute destinations that define the minimal motion across all windows without missing a possible solution.) Currently, if a solution is not found, then some of the rectangles remain in place despite overlap.



(a)



(b)



(c)

```
Method getLargestSpacewithAspectRatio(float aspectRatio){
   Rectangle largestSpace = null
   float area, largestArea = 0
   for each empty-space O in S.largestEmptySpaceList {
      if (O.height * aspectRatio >= O.width)
         /* use width to determine area */
         area = O.width / aspectRatio
      else
         /* use height to determine area */
         area = O.height * aspectRatio
      if (area > largestArea){
         largestArea = area
         largestSpace = O
      }
   }
   return (largestSpace)
}
```

(d)

Figure 13: Space manager applied to automated information visualization system. (a) Folders on timeline contain data about patient's operation. (b) Visualization system uses space manager to position and scale selected folder to reach largest size possible at its original aspect ratio without overlapping important objects. (c) Space manager debugging tool shows dark full-space rectangles and edges bounding empty spaces used to calculate enlarged folder. Gridded timeline background and selected folder in (a–b) are deleted from spatial representation by visualization system to allow scaled folder to overlap them. Large light gray rectangle at right is largest empty-space rectangle selected by query to contain scaled folder. (d) Pseudocode for query that selects largest empty space. Allocation strategy must also determine where to place object in selected largest empty-space rectangle.

Rather than adjust window positions only after a dragged window has been dropped, we can instead allow the motion to occur dynamically during the move itself. For example, while the dragged window is moved, each overlapped window can be moved out of the way.

In complementary work, Badros, Nichols, and Borning have explored the incorporation of modern constraint systems within a programmable window manager [1]. Their SCWM (Scheme Constraints Window Manager) allows the user to place classical constraints on windows; for example, to keep one window to the left of another or to keep the total height of two windows constant. While these constraints can allow interesting and useful behavior, they operate only on the representations of the windows (full-space rectangles). Thus, while a window can push a chain of windows as it moves, it cannot move them to prevent overlap with other windows because the system has no useful representation of the empty-space. We believe that an empty-space representation, such as that described here, would be an especially useful addition to such a constraint-based window manager.

6.2 Space Manager for 3D Presentation Layout

We have incorporated our space manager into IMPROVISE, an existing research system that automatically generates 3D information visualizations [16]. IMPROVISE's hierarchical-decomposition partial-order planner maps a set of communicative goals to 3D presentation objects. While this system also uses a constraint solver [6] to constrain object geometry, it originally used a space representation that included only the objects being laid out.

Figure 13(a) shows how IMPROVISE displays a timeline that presents a patient's surgery. Each of the folders in this interface contains information about the patient that the user may wish to explore in more detail. Enlarging a folder

to fill the display or popping it up without regard to overwriting important objects in the scene can cause a disturbing loss of context. Instead of creating a single area dedicated solely to the display of enlarged objects, we have used the space manager to allow the system to query the available space to find a suitable largest empty-space rectangle. For example, the query used to create Figure 13(b) finds the empty-space rectangle that can contain the largest possible rendition of the enlarged object at its original aspect ratio.

Each of the full-space rectangles in this application is the upright bounding box of the 2D projection of its associated 3D object. The ability to delete objects from the representation is an important feature. For example, the system specifies that the objects in the time chart must remain visible, while the large gridded time chart background and the area originally occupied by the selected folder need not be visible. Therefore, as shown in the debugging display of Figure 13(c), the system temporarily deletes the time chart's background and the original folder from the space representation that is used to calculate the geometry of the enlarged folder. This allows the folder to overlap the background and folder's original position, while not overlapping the time chart's contents. Figure 13(d) shows the query written by the space manager's user to select the largest empty-space rectangle in which to display the enlarged folder; an alternative query might find the empty-space rectangle closest to the folder's original location that can accommodate some specified minimum scale factor.

7. CONCLUSIONS AND FUTURE WORK

We have presented an approach to representing both full and empty 2D space in user interfaces, and have demonstrated how it may be used with several example applications. A number of variations on our system are possible. Some of these involve the way in which the system is used, without modifying the system itself:

- *Approximated coordinates.* The precise floating-point representation can potentially result in a number of extremely narrow or short largest empty-space rectangles, or frequent updates to the representation (e.g., if the full-space rectangles represent 2D projection-plane extents of tracked 3D objects). One solution is for the user to expand all full-space rectangles added so that their edges are on boundaries that are some multiple of user-chosen constant *N*. These discretized approximations would insure that no space is created that has a width or height smaller than *N*.

- *Approximations to hierarchy.* Since the space manager is a Java object, the user can support hierarchical containment by creating a number of space managers. Each of the space managers would then be treated by the user as managing a set of objects associated with a containing object at a higher level of the hierarchy. Alternatively, recall that any full-space rectangle wholly contained within another full-space rectangle has no effect on the empty-space representation, regardless of the order in which the contained and containing objects were added. Thus, if the user adds a full-space rectangle *F* that completely covers a set of full-space rectangles already in the representation, the largest empty-space rectangles that were contained within the area of *F* are deleted from the empty-space representation (although the full-space rectangles remain). Removing *F* will add any largest empty-space rectangles covered by *F*. Thus, adding and deleting a covering rectangle can be used to approximate some of the computational benefits of a true hierarchy.

- *Unbounded workspace representation.* The space manager is currently initialized by the user to work within a bounded rectangular workspace. Alternatively, to support infinitely zoomable user interfaces [2], it can be used as an unbounded manager, whose workspace ranges from $-\infty$ to $+\infty$ in each dimension (e.g., represented as the largest negative and positive floating point values). When computing window dimensions during a query, the user could optionally clip these to her own, possibly changing, finite min/max bounds.

Other variations involve modifications to the space manager itself that we are currently investigating:

- *Efficient batch add.* While the current incremental add is well suited for incremental additions, there are situations in which multiple objects will need to be added at known locations. For example, the deletion of a single rectangle *F* currently requires the creation of a subset space manager that incrementally adds all full-space rectangles intersected by *F*. We are in the process of designing a plane sweep algorithm that we believe could make possible a more efficient batch add algorithm, and consequently improve the efficiency of a delete.

- *Extension to 3D.* Our current 2D space manager has been applied to 3D only insofar as it addresses upright projection plane bounds. Since our algorithms treat each dimension separately, due in part to the use of axis-aligned rectangles, we are extending it to three or more dimensions. This would allow its use for computing available space in graphics applications that rely on 3D axis-aligned cuboid bounds.

- *Modifying objects*. Implementing an explicit object geometry modification function, or implicitly recognizing groups of deletes and adds that effectively modify a set of objects, could result in increased

efficiency. For example, when changes in the positions or sizes of a set of full-space objects are sufficiently small, changes needed in the empty space representation are minimal, involving only changes in the coordinates of edges, rather than the addition and removal of largest empty spaces currently caused by a delete/add pair. Similarly, suppose that a modified object's new geometry intersects its original geometry, and the intersection covers a large number of other objects. To avoid processing these other objects during the deletion and addition, the portion of the original geometry outside the intersection may be deleted and the portion of the new geometry outside the intersection added, leaving the intersection untouched.

ACKNOWLEDGMENTS

This work was supported in part by the National Library of Medicine Grant 5-R01 LM06593-02; the New York Science and Technology Center for Advanced Technology in Information Management, Computer Science, Medical Informatics and Genomics; Office of Naval Research Contracts N00014-97-1-0838, N00014-99-1-0249, and N00014-99-1-0394; and by equipment gifts from Intel and Microsoft. Thanks to Michelle Zhou for developing the IMPROVISE system to which we added the space manager, and to Steve Nowick for helpful discussions on the application of the consensus theorem to the computation of largest empty-space rectangles.

REFERENCES

[1] Badros, G., Nichols, J., and Borning, A. SCWM: An Extensible Constraint-Enabled Window Manager. *Proc. AAAI 2000 Spring Symposium on Smart Graphics*, March 20–22, 2000, Stanford, CA (http://scwm.mit.edu).

[2] Bederson, B., Hollan, J., Perlin, K., Meyer, J., Bacon, D., and Furnas, G. Pad++: A Zoomable Graphical Sketchpad for Exploring Alternate Interface Physics. *Jnl. of Visual Languages and Computing*, 7, 3–31, 1996.

[3] Bernard, M., and Jacquenet, F. Free space modeling for placing rectangles without overlapping. *Jnl. Universal Comp. Sci.,* 3 (6), 1997, 703–720.

[4] Clark, J.H., Hierarchical Geometric Models for Visible Surface Algorithms, *CACM*, 19(10), October 1976, 547–554.

[5] Cohen, E., Smith, E., and Iverson L. Constraint-Based Tiled Windows. *IEEE Comp. Graph. & Applics*., 6 (5), May 1986, 35–45.

[6] Gleicher, M. and Witkin, A. Snap Together Mathematics, Blake, E. and Wisskirchen, P. (eds.), *Proc. Eurographics Workshop on Object-Oriented Graphics*, Springer-Verlag, 1990, 21–34.

[7] Java: http://java.sun.com

[8] Kandogan, E. and Shneiderman, B. Elastic Windows: Evaluation of Multi-Window Operations, *Proc. CHI '97*, ACM, New York, (1997), 29–38.

[9] Katz, Randy H. *Contemporary Logic Design.* Benjamin Cummings/Addison Wesley Publishing Company, Redwood City, CA, 1994.

[10] Kay, T. and Kajiya, J. Ray Tracing Complex Scenes, *Comp. Graphics* (*Proc. ACM SIGGRAPH '86*), 20 (4), August 1986, Dallas, TX, 269–278.

[11] Ousterhout, J. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Trans. on Comp.-Aided Design of Integrated Circuits & Sys., CAD-3* (1), January 1984, 87–100.

[12] Ousterhout, J., Hamachi, G., Mayo, R., Scott, W., and Taylor, G. The Magic VLSI Layout System. *IEEE Design & Test of Computers*, 2 (1), February 1985, 19–30.

[13] Samet, H. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1990.

[14] Scheifler, R. and Gettys, J. The X Window System. *ACM Trans. on Graphics*, 5 (2), 1986, 79–109.

[15] Teitelman, W. A Tour Through CEDAR. *IEEE Software*, 1 (2), April 1984, 44–73.

[16] Zhou, M. and Feiner, S. Visual Task Characterization for Automated Visual Discourse Synthesis. *Proc. CHI '98* (*Conf. On Human Factors in Computing Systems*), Los Angeles, CA, April 18–23, 1998, p. 392–399.